ENPM818Z: Assignment #1
Title: LiDAR–Camera Fusion for 3D Object Detection
Due Date: October 13, 2025 **Points**: 30 pts

v1.0

Contents

1 Changelog

- v1.0: Original version.

2 Guidelines

This assignment must be completed by **groups of 3–4 students**. Follow all instructions carefully.

- Your solution must be original. Sharing code between groups or using external solutions is prohibited.
- Use a private GitHub repository (or equivalent) for version control. Public repositories are not permitted.
- Submit one compressed archive (.zip) per group via Canvas. Include all source code, your final report, and a short video.
- AI tools (e.g., ChatGPT, Copilot) may be used for boilerplate, debugging, and concepts. You must understand, cite, and be able to explain every line of submitted code.
- Provide a README.md with exact run instructions, including a one-line command to execute your pipeline.
- Your implementation should be reproducible by TAs without modification (assuming dataset and dependencies are prepared).

3 Introduction & Objectives

This assignment focuses on a cornerstone of autonomous vehicle perception: **sensor fusion**. You will build a camera–LiDAR fusion pipeline for 3D object detection (critical for safe navigation), tackling sensor calibration, projection, frustum culling, and 3D box estimation.

Cameras provide texture and color but not reliable depth; LiDAR provides geometry but no appearance. Fusing them yields robust, accurate scene understanding. We use the **KITTI Vision Benchmark** to ground your work in a real dataset.

4 Academic Context & Inspiration

The pipeline mirrors a classic approach in the literature. It is inspired by:

Frustum PointNets for 3D Object Detection from RGB-D Data
Charles R. Qi et al., CVPR 2018

A 2D detector proposes image boxes that are extruded into 3D frustums; LiDAR points inside each frustum are then used to estimate a 3D box. You will implement the core geometric pieces of this idea (projection, frustum culling, and 3D box estimation). **Paper link:** `https://arxiv.org/abs/1711.08488`

5 Recommended Dataset

Use the **KITTI Object Detection Benchmark**. It includes synchronized images, LiDAR point clouds, and calibration files; labels include 2D and 3D boxes.

- Download from: `http://www.cvlibs.net/datasets/kitti/eval_object.php?obj_benchmark=3d`
- Required zips (unzip under a single training directory):
  - data_object_image_2.zip → training/image_2 (left color images, .png)
  - data_object_velodyne.zip → training/velodyne (LiDAR point clouds, .bin)
  - data_object_calib.zip → training/calib (calibration, .txt)
  - data_object_label_2.zip → training/label_2 (2D/3D labels, .txt)

6 Assignment: LiDAR–Camera Fusion for 3D Object Detection

The project has four parts. You may use local Python scripts, Jupyter, or Colab.

6.1 Part A: Environment Setup & Data Exploration

Narrative overview. The first step is to set up your development environment and ensure you can correctly load and parse the KITTI dataset's complex structure. In real-world perception projects, build failures and data I/O issues consume a large fraction of time. This part intentionally front-loads environment reproducibility and data sanity checks. You will (1) create a clean, shareable Python environment, (2) verify your dataset layout and filenames, (3) implement or use the provided loaders for images, LiDAR .bin, and calibration .txt, and (4) produce simple visual confirmations that your inputs are correct. By the end of Part A you should have a reliable scaffold that your team can use for all subsequent parts without touching environment details again.

6.1.1 Environment managers (choose one)

Use `venv+pip`, `pip-tools`, `Poetry`, `uv`, or `conda`. Provide exact commands in README.md and a lockfile (requirements.txt or equivalent).

```
python3 -m venv .venv
source .venv/bin/activate      # Windows: .venv\Scripts\activate
pip install --upgrade pip
pip install numpy opencv-python matplotlib open3d
pip freeze > requirements.txt
```

6.1.2 Expected KITTI directory layout

```
training/
  image_2/      000000.png, 000001.png, ...
  velodyne/     000000.bin, 000001.bin, ...
  calib/        000000.txt, 000001.txt, ...  (e.g., 000145.txt)
  label_2/      000000.txt, 000001.txt, ...
```

Tip: zero-pad indices to 6 digits, e.g., `f"{idx:06d}.png"`.

6.1.3 Data loading: helper APIs and shapes

- `load_image(path)` → `H×W×3 uint8 (RGB)` (OpenCV BGR → RGB).
- `load_velodyne_bin(path)` → `N×4 float32` (columns: x, y, z, reflectance).
- `load_calib_file(path)` → `dict` with keys `"P2"` (3×4), `"R0_rect"` (3×3), `"Tr_velo_to_cam"` (3×4).

Print shapes once to confirm.

6.1.4 Initial visualization (sanity)

- Show the RGB image (no axes).
- Render the raw point cloud in Open3D (gray). Verify ground and objects are visible.
- Optional: histogram of height z or range $\sqrt{x^2 + y^2 + z^2}$.

6.1.5 Deliverable for Part A

- Console log with shapes (image, points, matrices).
- Figure: image; figure: Open3D view (screenshot).
- One-line command in README.md to reproduce for a given `--idx`.

6.2 Part B: Sensor Calibration and Projection

Narrative overview. The goal of this section is to solve the core geometric challenge of sensor fusion: aligning the disparate coordinate systems of the LiDAR and the camera. Imagine the LiDAR sensor at the center of its own 3D world, and the camera at the center of another. Because these sensors are mounted at different physical positions and have different orientations on the vehicle, a 3D point (x, y, z) measured by the LiDAR has no inherent meaning in the camera's frame. A point that is "forward" for the LiDAR might be "up and to the right" from the camera's perspective.

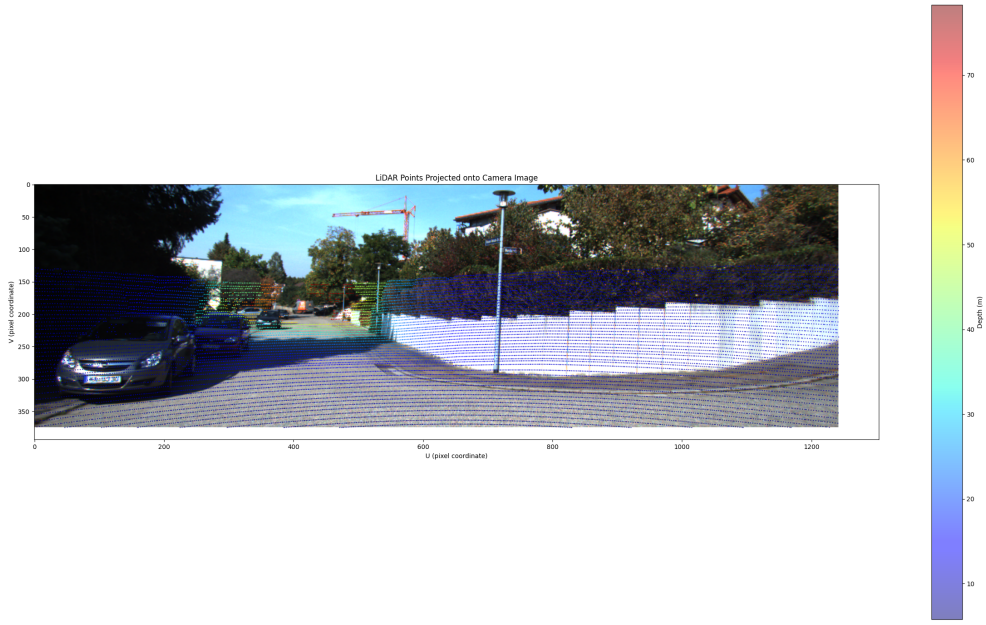

Figure 1: *Part B output for frame 250.*

To solve this, you will implement a multi-step mathematical pipeline using the provided calibration matrices. This process first applies a **rigid body transformation** (a rotation and a translation) to move the LiDAR point from its own coordinate system into the camera's 3D space. From there, a final **projection transformation** maps that 3D point onto the camera's 2D image plane, resulting in a specific (u, v) pixel coordinate. Mastering this transformation is the key to bridging the two distinct sensor views and allowing data from both to be correlated.

Expected Output: A robust projection function and a visualization that overlays projected LiDAR points on the image. Correct projections align with visible objects and road markings. Figure 1 shows the output for frame 250.

6.2.1 Tasks

- **T0 – Parse calibration (once per frame).**
  1. Read the `calib/{idx}.txt` file as plain text.
  2. Extract these keys and reshape:
     - `P2` → reshape to (3×4).
     - `R0_rect` → reshape to (3×3).
     - `Tr_velo_to_cam` → reshape to (3×4).
  3. Pad to homogeneous form (needed for matrix chaining):

$$R^h_{0\_\text{rect}} = \begin{bmatrix} R_{0\_\text{rect}} & 0 \\ 0^\top & 1 \end{bmatrix} \in \mathbb{R}^{4\times4}, \qquad T^h_{\text{velo}\to\text{cam}} = \begin{bmatrix} \text{Tr\_velo\_to\_cam} \\ 0^\top \quad 1 \end{bmatrix} \in \mathbb{R}^{4\times4}.$$

  4. Sanity print once (shapes only): `P2=(3,4)`, `R0_rect=(3,3)`, `Tr_velo_to_cam=(3,4)`.
- **T1 – Prepare data (per frame).**
  1. Load image `image_2/{idx}.png` as RGB. Let image width/height be (W, H).
  2. Load LiDAR `velodyne/{idx}.bin` as an $N\times4$ array: columns $(x, y, z, \text{reflectance})$ in the LiDAR frame.
  3. Form homogeneous LiDAR points:

$$X^h_{\text{velo}} = \begin{bmatrix} x & y & z & 1 \end{bmatrix}^\top \quad \Rightarrow \quad X^h_{\text{velo}} \in \mathbb{R}^{4\times N} \text{ (stack all } N \text{ points column-wise).}$$

- **T2 – Transform LiDAR → camera (rectified).**

1. Apply the rigid transform to the camera frame (unrectified):

$$X^h_{\text{cam}} \;=\; T^h_{\text{velo}\to\text{cam}}\, X^h_{\text{velo}}.$$

2. Apply rectification:

$$X^h_{\text{rect}} \;=\; R^h_{0\_\text{rect}}\, X^h_{\text{cam}}.$$

3. Extract camera-frame depths from the rectified coordinates:

$$Z_{\text{cam}} \;=\; (X^h_{\text{rect}})_{3,:} \text{ (row index 3 in 0-based, i.e., the } z \text{ row).}$$

4. **Depth filter:** keep only points with $Z_{\text{cam}} > 0$. Build a boolean mask and apply it to X^h_{rect} (and keep it to filter other arrays consistently).

- **T3 – Project to pixels (camera image plane).**
  1. Apply the projection matrix:

$$Y \;=\; P_2\, X^h_{\text{rect}} \in \mathbb{R}^{3\times N}.$$

  2. De-homogenize to pixel coordinates:

$$u \;=\; \frac{Y_{1,:}}{Y_{3,:}}, \qquad v \;=\; \frac{Y_{2,:}}{Y_{3,:}}.$$

  3. **Image bounds filter:** keep points with $0 \le u < W$ and $0 \le v < H$. Combine with the depth mask from the previous step.
  4. **What to keep for later:** (u, v) as `float`, Z_{cam} (depth per point), and the index mask of valid projected points.
- **T4 – Build the projection function (one clean API).**
  - **Signature (suggested):**

$$\texttt{project\_lidar\_to\_image(pts\_Nx4, calib)} \;\to\; \begin{cases} \texttt{uv\_Mx2 (float)} \\ \texttt{zcam\_M (float)} \\ \texttt{mask\_N (bool)} \end{cases}$$

  - **Do not filter** inside this function when projecting 3D box corners; instead, provide an option (e.g., `filter_points=False`) to skip masking so all 8 corners project (even if some are off-image).
- **T5 – Validate visually.**
  1. Plot the image (`matplotlib.imshow`) and overlay a scatter of (u, v) for valid points.
  2. Color points by depth (recommended: `c=Z_cam`, `cmap='jet'`, small marker size, alpha ≈ 0.5).
  3. **What "correct" looks like:** road points align with the road in the image; projected points concentrate on visible cars, poles, and buildings.

6.3 Part C: 2D Detection and 3D Data Association

Narrative overview. This part exploits complementary sensing: LiDAR provides precise geometry while the camera localizes object regions. Your goal is to associate each 2D image region with its corresponding subset of LiDAR points using **frustum culling**. A 2D detection acts like a cookie-cutter in the image plane; by projecting all LiDAR points and selecting those that fall inside the 2D region (and lie in front of the camera), you isolate a 3D cluster for that object. This cluster is the input to Part D.

Expected Output: A mapping from each 2D object region to a LiDAR point subset. Visualize (1) the 2D box overlaid on the image, and (2) a 3D scatter showing only the associated points. Figure 2 shows the output for frame 250.

6.3.1 Tasks

- **T0 — Choose a source of 2D regions.**
  - **Track A (recommended):** Use KITTI `label_2` "Car" boxes or the provided detector outputs. No detector setup required.
  - **Track B (optional):** Build simple proposals by clustering projected LiDAR points in the image plane (e.g., with DBSCAN) and taking their 2D bounding boxes.

Figure 2: *Part C output for frame 250.*

- **T1 — Project the LiDAR once (per frame).**
  - Use your Part B function to get the projected pixel coordinates (u, v) and camera-frame depths Z_{cam} for all LiDAR points.
- **T2 — Frustum Culling.**
  - For each 2D box $[u_{\min}, v_{\min}, u_{\max}, v_{\max}]$, create a boolean mask to select all LiDAR points that project inside it.

$$\mathbb{1}_{\text{box}} = (u_{\min} \le u \le u_{\max}) \wedge (v_{\min} \le v \le v_{\max}).$$

  - Combine this with a depth filter ($Z_{\text{cam}} > 0$). The 3D points corresponding to this final mask form the object cluster.
- **T3 — Depth Gating (optional but recommended).**
  - To remove background points (e.g., a wall behind a car), further refine your cluster.
  - Calculate the median depth $\tilde{Z}$ of the points within the cluster.
  - Keep only the points whose depth Z_i is within a certain range of the median, e.g., $Z_i \in [\tilde{Z} - \Delta, \tilde{Z} + \Delta]$, where Δ could be 3-5 meters.
- **T4 — Visualize the Association.**
  - On the 2D image, draw the 2D detection box.
  - In a 3D viewer, display *only* the final cluster of 3D points associated with that box.

6.4 Part D: 3D Bounding Box Estimation & Visualization

Narrative overview. You now have isolated 3D clusters per object. Raw point sets are not ideal for planning or tracking, which need a compact description: position, size, and orientation. In this part you turn each cluster into a **3D bounding box**. An axis-aligned box (AABB) is a simple, fast baseline. An oriented box (OBB) better matches real vehicle headings and yields tighter fits. You will compute these boxes and visualize them both in 3D and projected back onto the 2D image to demonstrate end-to-end consistency.

Expected Output: For each detection from Part C, produce a 3D box and two visualizations: (1) 2D image with the wireframe of the projected 3D box over the detection, (2) 3D scene with boxes around clusters. Figure 3 shows the output for frame 250.

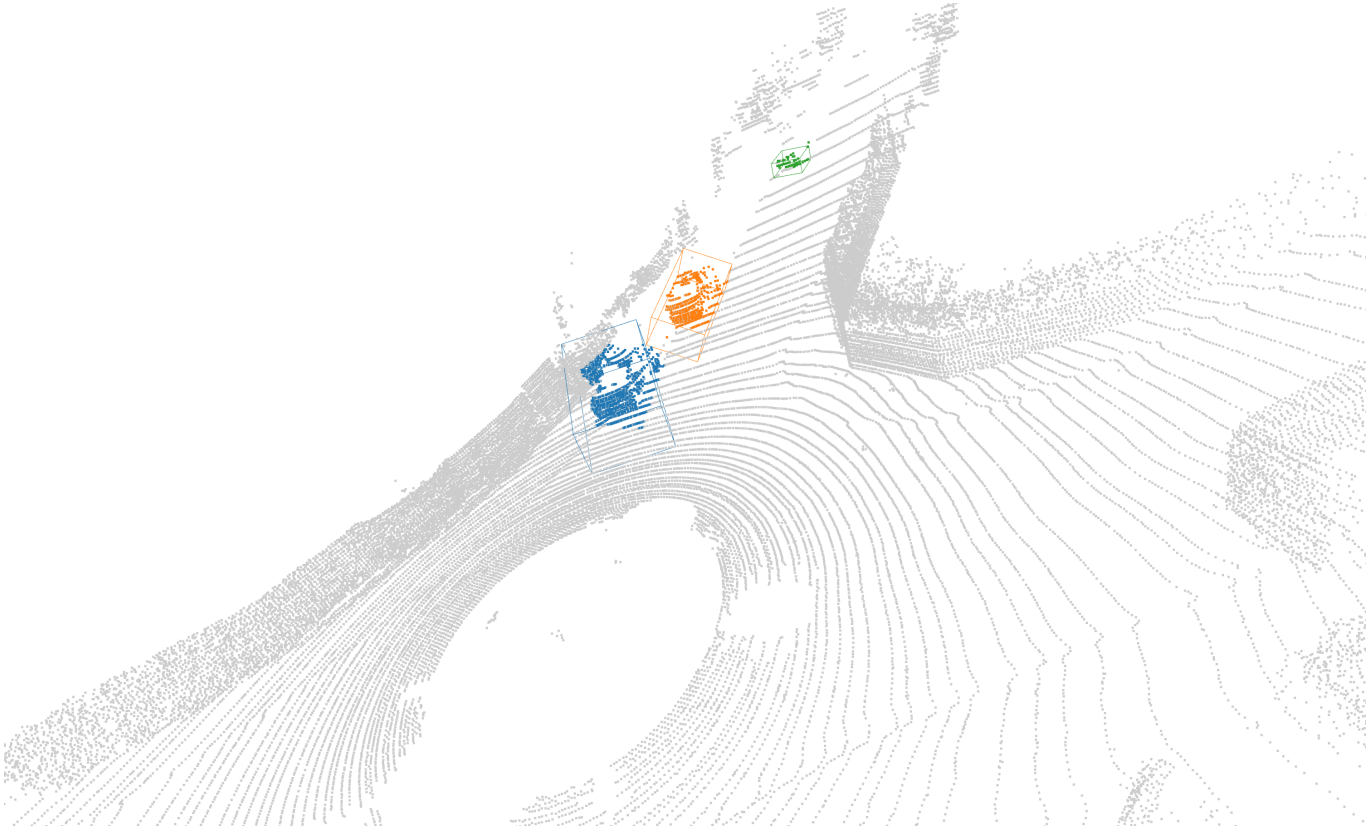

Figure 3: *Part D output for frame 250.*

6.4.1 Tasks

- **Task 1: Input and Pre-processing**
  - Choose a consistent reference frame for this part (**camera frame** is recommended).
  - For each point cluster from Part C, perform light clean-up (e.g., remove outliers) and ensure you have a minimum number of points (e.g., 40).
- **Task 2: Estimate 3D Box Parameters**
  - Your goal is to compute the center, dimensions, and orientation for each object cluster.
  - **AABB (Required Baseline):**
    - Find the coordinate-wise min/max of the points to define the box corners.
    - From these, calculate the **center** and **dimensions** (l, w, h).
  - **OBB (Bonus via PCA):**
    - Compute the point cloud's **centroid** (this is the box center).
    - Perform PCA on the centered points to find the three principal axes. These form the columns of the rotation matrix R.
    - Project the points into this new coordinate system to find the min/max extents, which define the box **dimensions** (l, w, h).
    - Extract the **yaw angle** θ by projecting the primary axis onto the ground plane.
- **Task 3: Finalize and Report Box Parameters**
  - For each detected object, consolidate its parameters into a clear format. State your chosen reference frame (e.g., "camera frame") in your report.
  - **Required Output:**
    - **'type'**: "AABB" or "OBB"
    - **'center'**: (x, y, z) in meters.
    - **'dims'**: (l, w, h) in meters.
    - **'yaw'**: θ in radians (set to 0 for AABB).
    - **(Optional)** A confidence metric (e.g., number of inlier points).
  - *Example JSON Output:*

```
{
  "frame": "camera",
  "type": "OBB",
  "center_m": [9.42, 1.52, 23.87],
```

```
    "dims_m": {"l": 4.23, "w": 1.87, "h": 1.62},
    "yaw_rad": 0.31,
    "n_points": 152
  }
```

- **Task 4: Visualization**
  - Create two key visualizations to validate your end-to-end pipeline.
  - **1. 2D Image View:**
    - For each 3D box, get its 8 corner points.
    - Project these 8 corners back onto the 2D image using your Part B function. **Important:** Temporarily disable filtering inside your projection function so that corners outside the image view are not dropped.
    - Draw the wireframe of the projected 3D box over the original 2D detection.
  - **2. 3D Scene View:**
    - In a 3D viewer (like Open3D), display the entire scene's point cloud.
    - For each detected object, render its 3D bounding box (AABB or OBB) around its associated point cluster. Use distinct colors for each box.

7 Student Notes & Frequently Asked Questions

- **Why is P_2's last column nonzero?** Stereo rectification uses $[K \mid Kt]$. For monocular projection, still use the full 3×4 P_2.
- **Which camera matches `image_2`?** The left color camera. Use P_2.
- **Depth filtering:** Always after transforming to the camera frame: require $z_{\text{cam}} > 0$.
- **Image coordinates:** Origin top-left; u right, v down. Matplotlib may need inverted y-axis.
- **Homogeneous padding:** Use $R^h_{0\_\text{rect}}$ and $T^h_{\text{velo}\to\text{cam}}$ as 4×4.
- **Common pitfalls:** Wrong P matrix, reversed matrix order, forgetting (de-)homogenization, mixing frames across steps, dropping 3D corners due to filtering inside the projector.
- **OpenCV vs. Matplotlib:** Convert BGR↔RGB if mixing drawing libraries.

8 Scope, Timeline, and Scaffolding

Designed to be challenging yet achievable in two weeks with good planning (**30–40 group-hours** total).

8.1 Scope Guardrails

- Process 50–100 frames (e.g., sequence 0000, frames 0–99).
- Focus on **Cars** only.
- Use KITTI boxes or provided detector outputs (a full detector is not required).
- AABB is required. OBB via PCA is **bonus**.
- Visualize one 3D scene and a grid of 8–12 representative images with projected 3D boxes.

8.2 Recommended Timeline

- **Week 1:** Finish data loading (Part A) and projection (Part B). Validate against the golden example.
- **Week 2:** Complete Part C (frustum culling) and Part D (box estimation & visualizations). Finalize report, README, and video.

8.3 Minimal Scaffolding Provided

- **Starter I/O functions:** Loaders for KITTI images, .bin, and calib .txt with a clean API. starter_code.py can be found on Canvas.
- **Matrix reference sheet:** Mapping for P_2, R_{0_rect}, $T_{\text{velo}\to\text{cam}}$, with shapes and the projection chain. The example below is from `000145.txt`
  - **LiDAR (Velodyne):** x forward, y left, z up.
  - **Camera:** x right, y down, z forward.
  - **Image:** Pixel origin at top-left; u increases rightward, v increases downward.

- **Camera Projection Matrix for `image_2`** The projection matrix for `image_2` is $P_2 \in \mathbb{R}^{3\times4}$, which maps rectified camera-frame 3D points to pixels:

$$P_2 = \begin{bmatrix} 707.0493 & 0 & 604.0814 & 45.75831 \\ 0 & 707.0493 & 180.5066 & -0.3454157 \\ 0 & 0 & 1 & 0.004981016 \end{bmatrix}$$

  The intrinsics embedded in P_2 are f_x=707.0493, f_y=707.0493, c_x=604.0814, c_y=180.5066. *Why is the 4th column nonzero?* KITTI encodes rectified stereo as $[K \mid K\,t]$; the last column relates to the stereo baseline. For monocular projection, still use the full 3×4 matrix as given.
- **Image Rectification** The rectification matrix is $R_{0\_rect} \in \mathbb{R}^{3\times3}$:

$$R_{0\_rect} = \begin{bmatrix} 0.9999128 & 0.01009263 & -0.008511932 \\ -0.01012729 & 0.9999406 & -0.004037671 \\ 0.008470675 & 0.004123522 & 0.9999556 \end{bmatrix}$$

  Pad to homogeneous form before chaining:

$$R^h_{0\_rect} = \begin{bmatrix} R_{0\_rect} & 0 \\ 0^\top & 1 \end{bmatrix} \in \mathbb{R}^{4\times4}.$$

- **LiDAR to Camera Rigid Transform** The rigid transform from Velodyne to the (unrectified) camera frame is $T_{\text{velo}\to\text{cam}} \in \mathbb{R}^{3\times4}$:

$$T_{\text{velo}\to\text{cam}} = \left[\begin{array}{ccc|c} 0.006927964 & -0.9999722 & -0.002757829 & -0.02457729 \\ -0.001162982 & 0.002749836 & -0.9999955 & -0.06127237 \\ 0.9999753 & 0.006931141 & -0.001143899 & -0.3321029 \end{array}\right]$$

  Homogeneous padding for chaining:

$$T^h_{\text{velo}\to\text{cam}} = \begin{bmatrix} T_{\text{velo}\to\text{cam}} \\ 0^\top \;\; 1 \end{bmatrix} \in \mathbb{R}^{4\times4}.$$

- **Other Projection Matrices** P_0, P_1, and P_3 map other (rectified) cameras. For `image_2`, always use P_2.
- **Full Projection Chain (LiDAR point to pixel)** Given a LiDAR point $x^h_{\text{velo}} = (x, y, z, 1)^\top$, compute

$$y_{\text{image}} = P_2\, R^h_{0\_rect}\, T^h_{\text{velo}\to\text{cam}}\, x^h_{\text{velo}}, \qquad (u, v) = \left(\tfrac{y_1}{y_3}, \tfrac{y_2}{y_3}\right),$$

  valid only if the camera-frame depth $z_{\text{cam}} > 0$ after applying $R^h_{0\_rect} T^h_{\text{velo}\to\text{cam}}$.
- **Practical Notes**
  - Filter by $z_{\text{cam}} > 0$ (camera frame), not LiDAR $x > 0$.
  - After projection, discard points outside the image window.
  - Typical shapes: P_2(3×4), R^h_0(4×4), T^h(4×4), x^h(4×1).
  - Keep the matrix order as written; do not commute terms.
  - When projecting 3D box corners, avoid filtering inside the projection helper (or you may drop vertices).
  - OpenCV draws in BGR; convert to/from RGB if mixing with Matplotlib.
- **Detector outputs (optional):** Pre-computed 2D boxes for your subset so you can skip running a detector. To use the provided 2D detections, you will need to parse the corresponding '.txt' label file for each frame. Each line in these files represents a single object. Below is a Python helper function to load these labels into a structured list of objects, along with an example of how to use it. This function specifically extracts the 'Car' objects and their 2D bounding boxes, which is all you need for Part C. detector.py can be found on Canvas.

9 Deliverables & Grading

Submit a single .zip containing:

- **Source code:** Python scripts or notebooks. Clean, commented, and reproducible on a different machine.
- README.md**:** Exact setup and a one-line command to run. Specify dataset paths and briefly describe each major script/notebook.
- **Demonstration video (MP4, 1–2 minutes):** Show (i) 2D images with projected 3D boxes and (ii) the 3D point cloud with boxes. Add short narration or on-screen text.

- **Short report (PDF, 2–4 pages):** A concise summary of your approach, results, and discussion. See the outline below for guidance.

9.1 Report Outline

Your report should be structured professionally and include the following sections.

- **1. Introduction (approx. 0.5 pages)**
  - State the problem: 3D object detection from camera and LiDAR data.
  - Briefly explain the objective of the assignment.
  - Summarize your technical approach at a high level (e.g., "We implemented a pipeline inspired by Frustum PointNets...").
- **2. Methodology (approx. 1-1.5 pages)**
  - **Sensor Projection:** Describe your implementation of the LiDAR-to-camera projection, including the transformation chain.
  - **Data Association:** Explain how you performed frustum culling to associate LiDAR points with 2D detections. Mention your choice of 2D boxes (e.g., ground truth) and describe your depth gating strategy.
  - **3D Box Estimation:** Detail your method for fitting Axis-Aligned Bounding Boxes (AABB). If you completed the bonus, also describe your Oriented Bounding Box (OBB) estimation using PCA.
- **3. Results (approx. 1 page)**
  - Present your key visualizations.
  - Include a figure with several examples of 2D images showing the initial 2D box and the final projected 3D wireframe.
  - Include a high-quality screenshot of the 3D scene visualization, showing the full point cloud with your estimated 3D boxes rendered.
  - Briefly comment on the quality and accuracy of your results in the figure captions or accompanying text.
- **4. Discussion & Conclusion (approx. 0.5 pages)**
  - Discuss any significant challenges you encountered during implementation.
  - Analyze the limitations of your pipeline. For instance, describe a scenario where it might fail (e.g., severe occlusion, distant objects).
  - Suggest one or two potential improvements for future work.
  - Conclude with a brief summary of your accomplishments.

9.2 Grading Rubric (30 points total + up to 2 bonus points)

The rubric below is the sole scoring reference.

- **Pipeline Implementation (22 pts):**
  - *Part A – Setup & Data Loading (3 pts):* Loads images, point clouds, and calibration files correctly.
  - *Part B – Calibration & Projection (7 pts):* Accurate projection, clear visual alignment, proper handling of $z_{\text{cam}} \leq 0$.
  - *Part C – 2D Detection & Association (6 pts):* Uses 2D boxes to perform frustum culling and isolate per-object clusters.
  - *Part D – 3D Box Estimation & Visualization (6 pts):* AABB implemented; final 2D and 3D visualizations are clear and correct.
- **Report & Deliverables (8 pts):**
  - *Report (4 pts):* Clear, well-structured, and covers all required sections of the outline.
  - *Demonstration Video (2 pts):* Succinct and complete presentation of results.
  - *Code Quality & README (2 pts):* Clean structure; one-line run command; reproducible.
- **Bonus (up to +2 pts):**
  - *Oriented Bounding Box (OBB) via PCA (+2 pts):* Correct PCA-based OBB, visualization, and a brief AABB vs. OBB comparison in the report.

10 Acronyms

- **AABB** — Axis-Aligned Bounding Box (box edges aligned with the chosen coordinate axes).
- **OBB** — Oriented Bounding Box (box oriented by object axes, e.g., via PCA).
- **PCA** — Principal Component Analysis (eigen-decomposition of covariance to get principal directions).
- **KITTI** — Karlsruhe Institute of Technology and Toyota Technological Institute dataset.